**KATHMANDU UNIVERSITY**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Dhulikhel, Kavre**


**A Project Report**

**on**

**"Spell Snake"**

**[COMP 310]**

**(For the partial fulfilment of 3rd year/1st Semester in Computer Science)**

**Submitted by:**

**Bigen Aryal (77)**

**Bikramaditya Subedi (75)**

**Milan Dhamala (73)**

**Rajeshwor Niroula (71)**

**Usta Adhikari (68)**

**Submitted to:**

**Department of Computer Science and Engineering**

**Submission Date: Aug 06, 2021**

# Bonafide Certificate

**This project work on**

**"Spell Snake"**

**is the bonafide work of**

**Bigen Aryal (77)**

**Bikramaditya Subedi (75)**

**Milan Dhamala (73)**

**Rajeshwor Niroula (71)**

**Usta Adhikari (68)**

**who carried out the work under my supervision.**

**Project Supervisor**

**Name: Dr. Rajani Chulyadyo**

**Academic Designation: Assistant Professor, DOCSE**

# Abstract

Classic snake and frog games are well-known among us. It is simple reasoning that is also quite useful for individuals learning about graphic APIs and getting into game creation. This classic has been replicated hundreds of times by young developers to learn the fundamentals of game programming, and we are no exception. However, the classic on its own is not very engaging, so we decided to add a twist, turning it into an instructive game. In “Spell Snake”, a grid of spread-out alphabets is shown where the user needs to gobble up those alphabets to form a word. This keeps repeating and resets when the snake strikes the boundary or wrong letter is eaten by the snake. The snakes keep growing with each alphabet being gobbled.

**Keywords: Spell, Snake, Alphabets, OpenGL, GLEW, GLFW**

# Table of Contents

## List of Figures

## Page No.

## List of Tables

## Page No.

# List of Acronyms

**API**: Application Program Interface

**OpenGL**: Open Graphics Library

**PC**: Personal Computer

**GLFW**: Graphics Library Framework

**GLM**: Generalized Linear Model

**GLSL**: OpenGL Shading Language

**GPU**: Graphics Processing Unit

**IDE**: Integrated Development Environment

**Px**: Pixels

**OS**: Operating System

**CPU**: Central processing unit

# Chapter 1: Introduction

'Spell Snake' is a Windows game that allows players to acquire the skill of spelling in an interactive and fun approach. Evolved from the golden age of classic arcade games, this game centres around a very known snake and frog game. The game is designed around a snake piloting on the screen and collecting the letters in a predetermined arrangement.

## 1.1. Background

Spelling, the art of correctly assembling words from their letters, is one of the essential components of successful writing. Being confident at spelling leads to confidence in all aspects of literacy. Teaching young spellers, the strategies, rules and concepts to grow their spelling and vocabulary knowledge benefits them in all aspects of their learning, as well as in their everyday life. In today's world, people are almost overtaken by various electronic devices like mobiles, laptops etc. Considering all these contexts we decided to develop this game as a motivational medium to learn and play on the same platform.

Being inspired by classic games, Project 'Spell Snake' is created using C++ as a programming language and OpenGL as a graphics library. OpenGL API has been in practice in making game engines such as Sauerbraten[1], Ballenger a Platformer etc. We used OpenGL due to its feature of rendering 2D and 3D vector graphics and a large set of functions that we can use to manipulate graphics and images. Spelling games if searched on the internet is not a unique game, but most of them are missing on efficiency, interaction and attractive visualization. 'Spell Snake' desktop application is easy to understand, interactive and fun with pleasing visualization.

## 1.2. Objectives

- To create a platform to learn spelling in a fun and interactive approach
- To give a classic game twist of an educational feature

[1] http://sauerbraten.org/

## 1.3. Motivation and Significance

This is a simple game that can be used to teach spellings to kids since this game provides a platform for them to enjoy and learn at the same time. Simple classics like snake and frog have become obsolete as a result of contemporary graphic technologies. Yet, for up-and-coming game developers, it might as well be their first game. We wanted to learn about graphics development while having fun, so we chose to turn the classic into a spelling game, which would make the game much more exciting and fruitful in the long run. Snake and frog is a simple game that is easy to learn, especially for children, and it may recover popularity with our twist.

# Chapter 2: Related Works

The following games resemble our work conceptually. These developments have been an insight to our project.

1) Cube 2: Sauerbraten: It is a cross-platform first-person shooter that runs on Windows, Linux and Mac OS X using OpenGL and SDL. The aim of the project is not to produce the most features and highest-quality graphics possible, but rather to allow map-editing to be done in real-time within the game, while keeping the engine source code small and elegant. (Lee Salzman, n.d.)

2) Snake and Frog: Snake and frog was first created as a concept in 1976 under the name of Blockage, and was a monochromatic two-player arcade game developed by video games company, Gremlin Interactive. With Taneli Armanto as a developer it was launched along with Nokia 6110 in 1997, at the time it literally defined the possibility of mobile gaming. (GamingHistory, 1976)

3) Hangman: Hangman is a one-player word guessing game. The user must suggest letters one by one once the algorithm guesses a word. The machine opens the word every time it contains a letter (all of them, if there are a few). When a word does not contain a letter, the computer penalises the user with a penalty point. The user loses if there are five penalty points. The computer loses if there are no hidden letters left. (Farlex, n.d.)

4) QuteMol: It is an open-source, interactive, molecular visualization system. QuteMol utilizes the current capabilities of modern GPUs through OpenGL shaders to offer an array of innovative visual effects. QuteMol visualization techniques are aimed at improving clarity and an easier understanding of the 3D shape and structure of large molecules or complex proteins. (Marco Tarini, n.d.)

These are just a sample; thousands of games can be found on the internet based on the snake concept. These snake games are available for both PC and mobile phones. Similar to our project, some of these games are developed for educational purposes too. Some of the snake game teach addition, subtraction, multiplication, division etc., where a player is given a certain problem and the player have to eat a frog that contains the solution.

# Chapter 3: Design and Implementation

In Spell Snake letters from random words are split and randomly scattered all over the screen. To improve the score, the snake needs to eat those letters in a sequence to make a word displayed in a status bar. If the word consumed is not in order, then the game is over but if all the letters ate are in the correct order, then another random word will be selected and the game goes on. The player is also not allowed to collide with the border or with the snake itself, in both cases the game is over. As the score improves snake becomes longer and the game becomes more challenging.

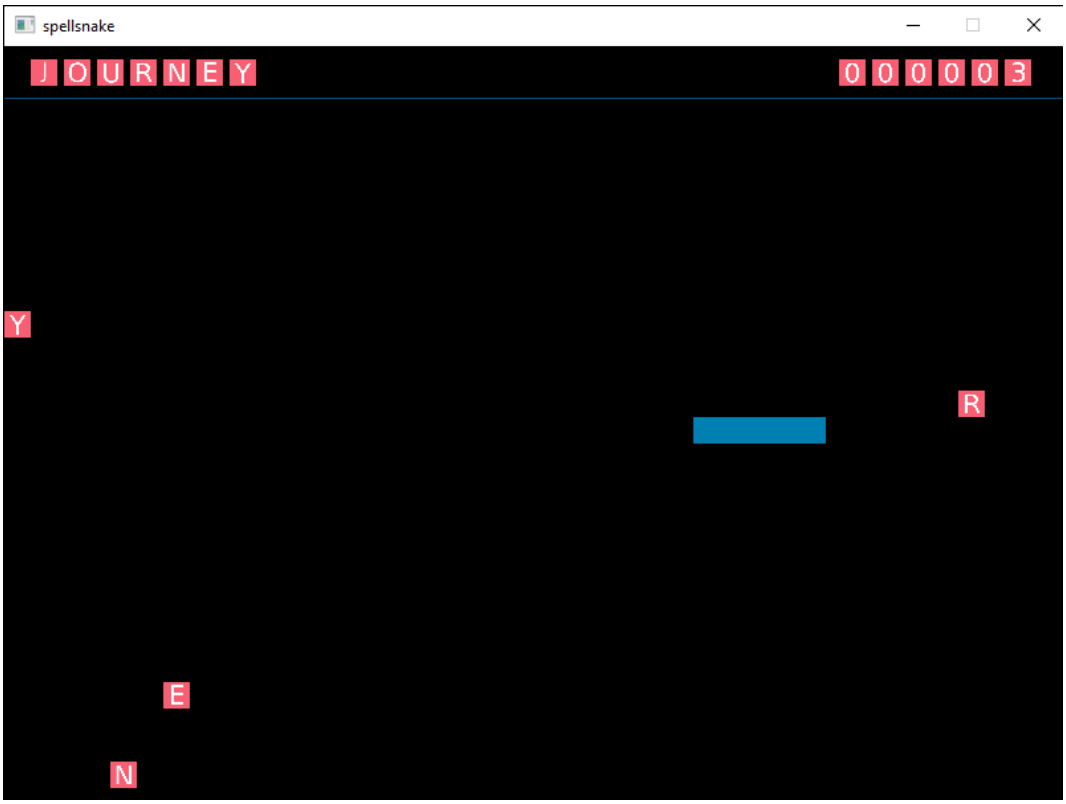


***Fig.3.1 Game view***

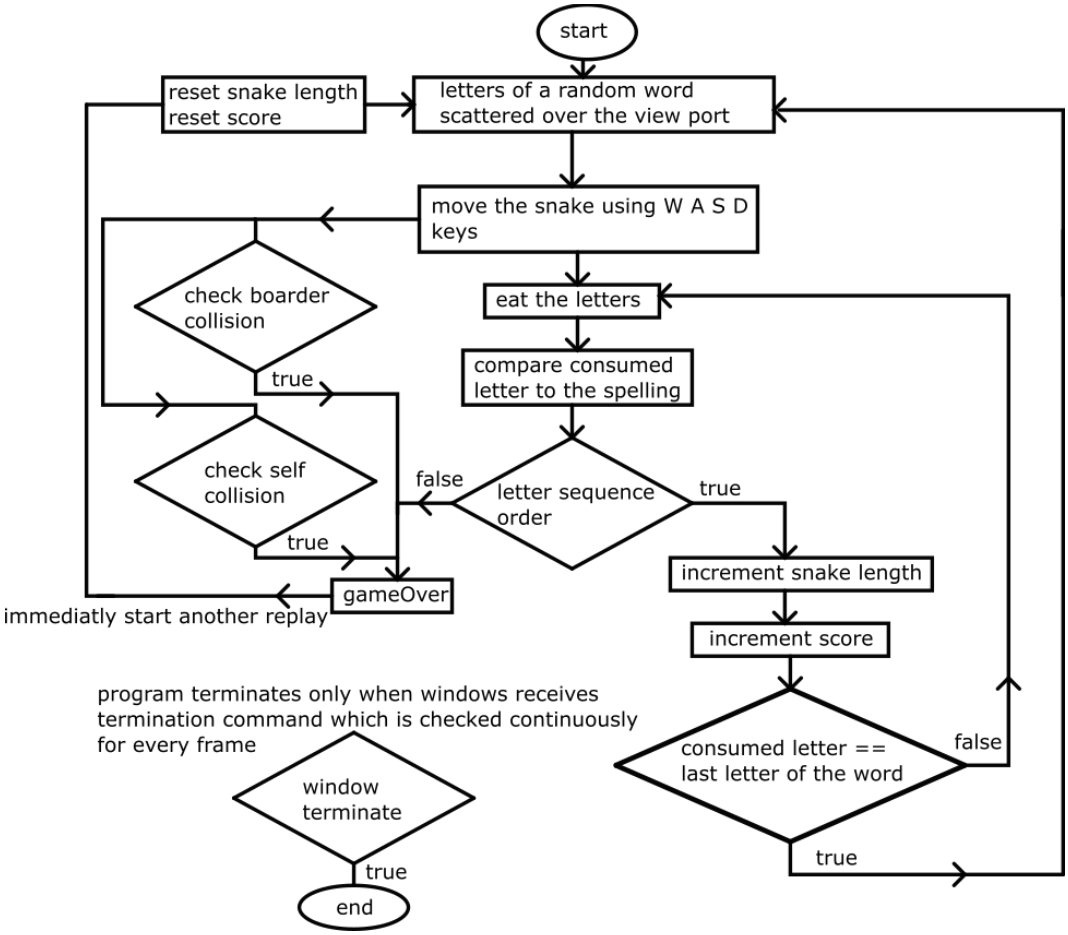


***Fig.3.2 Gameplay Design***

## 3.1. Structure of the main block:

Gameplay design mentioned in Fig.3.2 is executed to following main block structure which implements multiple classes and functions that are abstracted as different files outside of the main block, each function of the following flowchart steps is discussed in detail throughout the chapter.

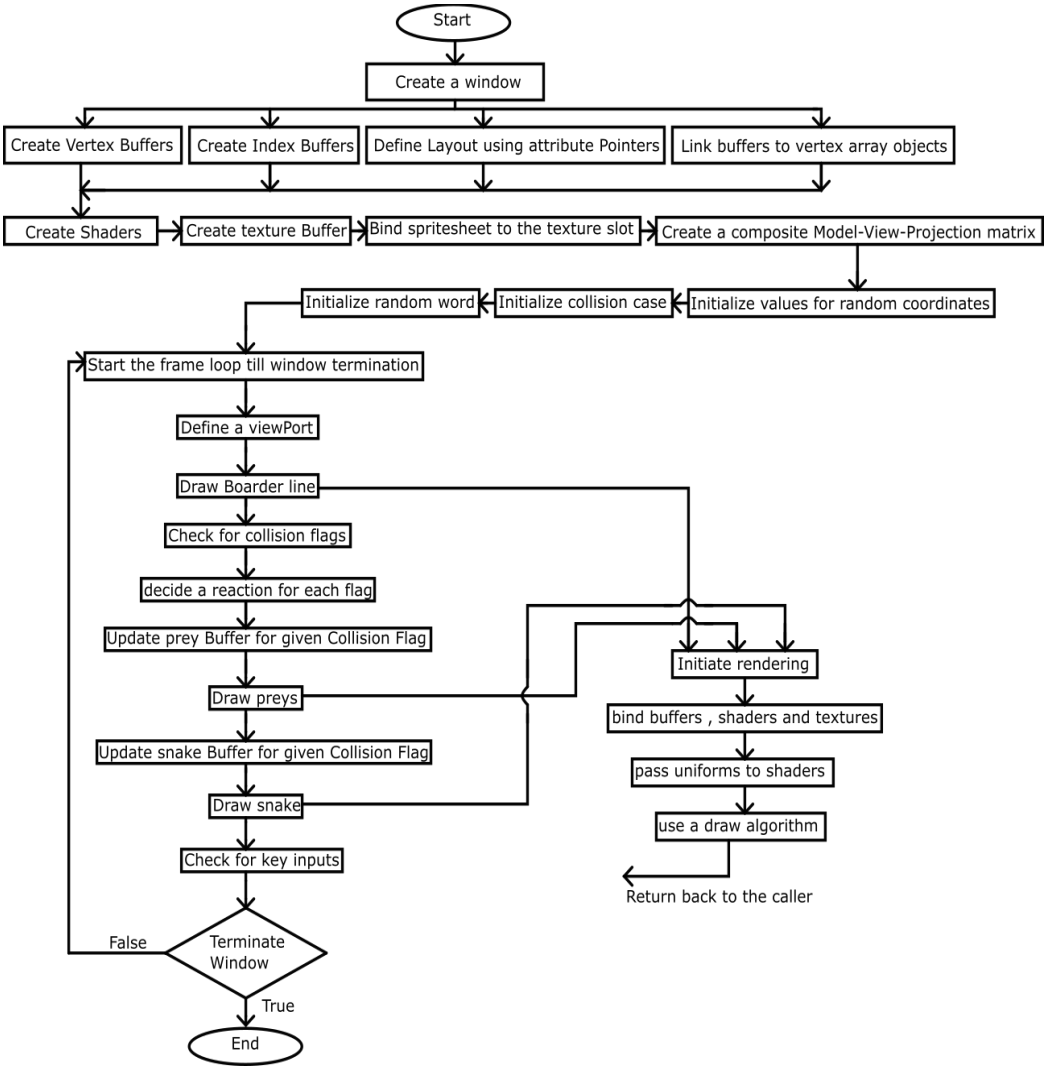


*Fig.3.1.1 Flow of the main block*

## 3.2 Transformation Pipeline

We have used the GLFW library for creating a fixed window of size of 800*600. The same window size is used for creating a viewport before any buffer is drawn on the screen it has to go through our transformation pipeline which we have achieved through the GLM math library and shaders.

```
glm::vec3 translationA(0,0,0);
glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(0, 0, 0));//view is constant for now so keeping it out of the loop
//making proj matrix and binding our shaders with uniforms
glm::mat4 proj = glm::ortho(0.0f, (float)screenWidth, 0.0f, (float)screenheight, -1.0f, 1.0f);
//in case if we use normalized coordinates glm::mat4 proj = glm::ortho(-ratio, ratio, -1.0f, 1.0f, -1.0f, 1.0f);
glm::mat4 model = glm::translate(glm::mat4(1.0f), translationA);
glm::mat4 mvp = proj * view * model; //remember this multiplication needs to happen in this order because opengl expects it
```

*Fig.3.2.1 MVP Matrix*

Once the model, view and projection matrices are created they are multiplied to form a composite transformation matrix “mvp” a vertex buffer will be processed by the vertex shader, where mvp will be multiplied to every single coordinate before they are passed to the fragment shader, the buffer is assigned with a certain texture selected from the sprite sheet according to the texture coordinates.

## 3.3 Shaders

Shaders are programs that are compiled and executed on a GPU (Graphics Processing Unit). It is an extremely powerful tool that allows us to implement various pipelines of rendering such as transformation, texture and colour binding on every pixel in every single frame displayed throughout the program cycle.

```
#shader vertex
#version 330 core

layout(location = 0) in vec4 position;
layout(location = 1) in vec2 textureCoordinate;

//out vec2 v_TextureCoordinate; //v stands for varying, simply we are outputting texture coordinates from vertex shader.
out vec2 fragCoordinate;

uniform vec2 v_TextureCoordinateShift;
uniform mat4 u_MVP;

void main()
{
 gl_Position =u_MVP * position;
 //v_TextureCoordinate = textureCoordinate;
 fragCoordinate = v_TextureCoordinateShift + textureCoordinate;

};
```

*Fig.3.3.1 Vertex Shader*

```
//remember fragment shader runs for every pixel

#shader fragment
#version 330 core

layout(location = 0) out vec4 color;

//in vec2 v_TextureCoordinate; //inputting texturecoordinates from vertex shader in fragment shader
in vec2 fragCoordinate;
uniform sampler2D u_Texture;

void main()
{
    //vec4 texColor = texture(u_Texture, v_TextureCoordinate);
    vec4 texColor = texture(u_Texture, fragCoordinate);
    color = texColor;
    //color = u_color;
};
```

*Fig.3.3.2 Fragment Shader*

Our shaders both vertex and fragment shader, are written using the GLSL language. These shaders are passed using a parseShader() function and passed to the shader compilation logic.

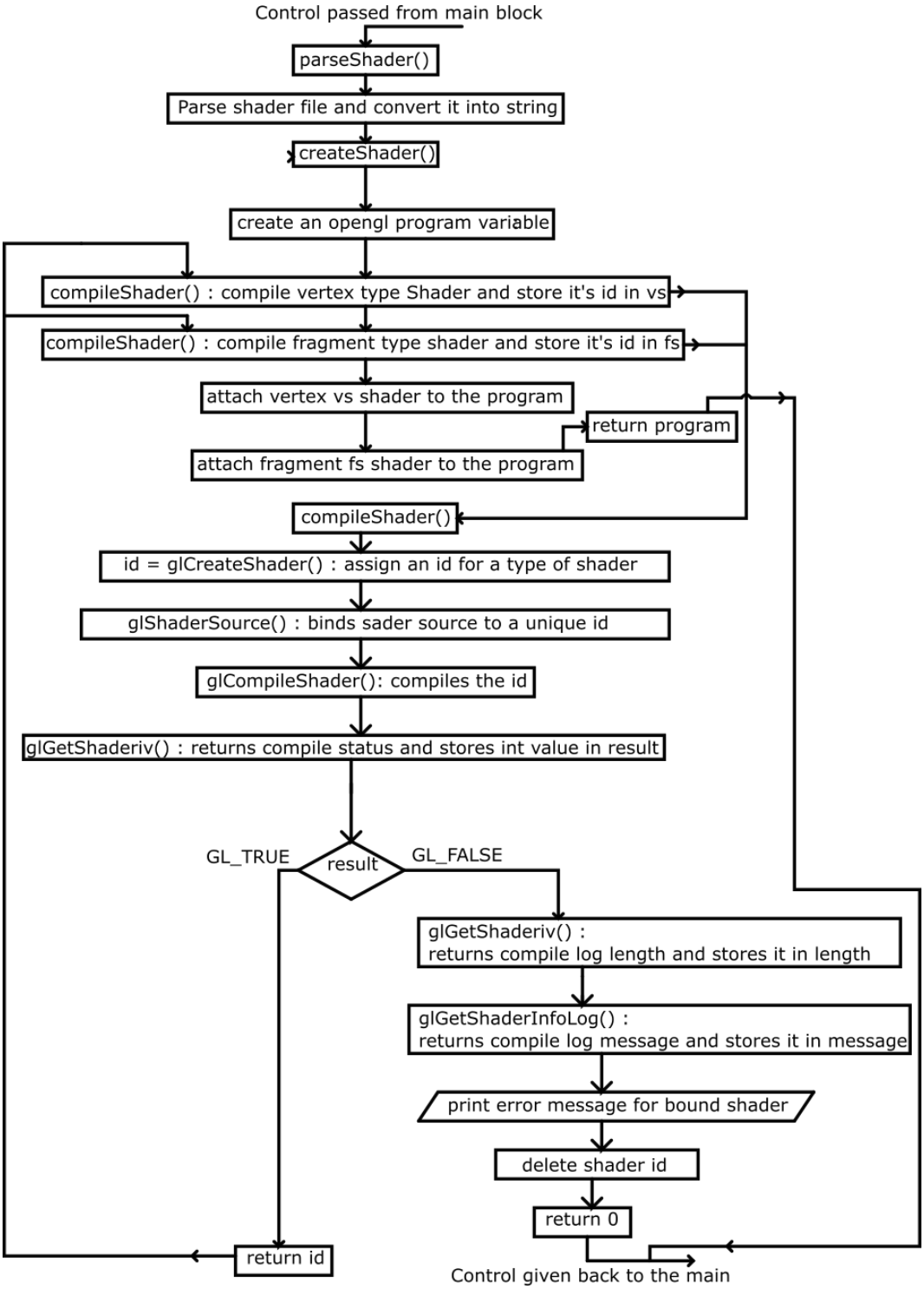


***Fig.3.3.3 Shader Compilation Logic***

Shader compilation includes two functions, compileShader() and createShader(). In compileShader() we assign a shader string source to an id and initiate compilation in GPU. Since the compilation happens in GPU our IDE compiler is not involved, hence we might not know of any error that might occur. So, we have also created a small error handling logic that tests compile status using glGetShader() and return an integer value. If the value is null, it means we have encountered an error and we redirect the error message to our IDE console by printing the compile log. For the compilation error, the program deletes the Shader-ID, else it will return ID to the function createShader().In createShader() we will create a program using function glCreateShader() and assign the returned shader id to the program and return the program to a caller that needs the shade implementation.

## 3.4 Buffers

The above discussed shaders are used by vertex buffers for rendering and rasterization. A code snippet of one of our buffers is shown in Fig.3.4.1.

```
positions[i+0] =x ;
positions[i+1] =y ;
positions[i+2] = spriteWidth * textureOffset;
positions[i+3] = 0.0f;
positions[i+4] = x+vertexOffset;
positions[i+5] = y;
positions[i+6] = (spriteWidth * textureOffset) + spriteWidth;
positions[i+7] = 0.0f;
positions[i+8] =x+vertexOffset ;
positions[i+9] = y+vertexOffset;
positions[i+10] = (spriteWidth * textureOffset) + spriteWidth;
positions[i+11] = 1.0f;
positions[i+12] = x;
positions[i+13] = y+vertexOffset;
positions[i+14] = spriteWidth * textureOffset;
positions[i+15] = 1.0f;

positions[i + 16] = statsDispX;
positions[i + 17] = statsDispY;
positions[i + 18] = spriteWidth * indexOrder[objCount];
positions[i + 19] = 0.0f;
positions[i + 20] = statsDispX + 20;
positions[i + 21] = statsDispY;
positions[i + 22] = (spriteWidth * indexOrder[objCount]) + spriteWidth;
positions[i + 23] = 0.0f;
positions[i + 24] = statsDispX + 20;
positions[i + 25] = statsDispY + 20;
positions[i + 26] = (spriteWidth * indexOrder[objCount]) + spriteWidth;
positions[i + 27] = 1.0f;
positions[i + 28] = statsDispX;
positions[i + 29] = statsDispY + 20;
positions[i + 30] = spriteWidth * indexOrder[objCount];
positions[i + 31] = 1.0f;
```

***Fig.3.4.1 A vertex buffer***

Each buffer consists of a position[ ] array where the first two coordinates represent position x and y respectively and the next two coordinates represent texture coordinates. The same sequence will continue till the end of the buffer placed within the loop. The only difference is the elements are incremented by required offsets to create geometry and to crop appropriate texture from the sprite sheet.

To create a square of twenty-pixel size for each side, the x and y coordinate has to be either incremented by 20 units or kept constant as per the coordinate we need.

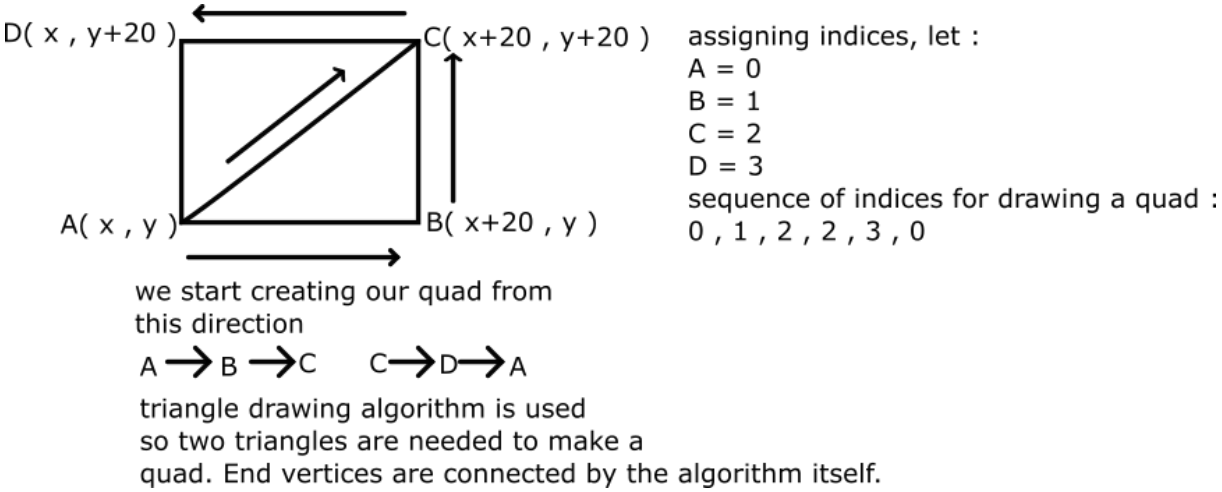


***Fig.3.4.2 A Quad Structure***

Since we have used a triangle drawing algorithm, we will need to create two triangles. To make a quad we will assign a unique index to each vertex of the quad and tell the algorithm to draw the quad with our sequence of indices as shown in the figure above. For multiple quads following algorithm was used for creating a sequence of indices.

```
static unsigned int indices[156];
unsigned int i = 0;
unsigned int offset = 0;
for (float y = 0; y < 26; y = y + 1)
{
        indices[i] = 0 + offset;
        indices[i + 1] = 1 + offset;
        indices[i + 2] = 2 + offset;
        indices[i + 3] = 2 + offset;
        indices[i + 4] = 3 + offset;
        indices[i + 5] = 0 + offset;

        offset = offset + 4;
        i = i + 6;
}
count = i;
return indices;
```

***Fig.3.4.3 Indices Sequence***

Later, attribute pointers were used to define the layout of the vertices and the layout was bounded to a unique vertex array object, which will be passed to the draw call along with index buffer and vertex buffer will be rendered and rasterized according to the logic of our bound shader.

```
//batching all 5 preys in one draw call
vbp.bind();
const void* preyVertexData = p.vertexData();// sizeOfData is calculated inside vertexData so we have to call vertexData befor using sizeOfData
glBufferSubData(GL_ARRAY_BUFFER, 0, p.sizeOfData(), preyVertexData);
shade.bind();
shade.setUniformMatrix4fv("u_MVP", mvp);
shade.setUniform1i("u_Texture", 0);
shade.setUniform2f("v_TextureCoordinateShift", 0.0 , 0.0);
render.draw(vap, ibp, GL_TRIANGLES, p.indexCountForCurrentLength());
```

***Fig.3.4.4 Binding elements before draw call***

```
void renderer::draw(const vertexArray& va, const indexBuffer& ib, unsigned int drawType, unsigned int indexCountCurrentCall) const
{
    //binding everything and using draw call
    va.bind();
    ib.bind();
    glCall(glDrawElements(drawType, indexCountCurrentCall , GL_UNSIGNED_INT, nullptr));

}
```

***Fig.3.4.5 Draw call***

## 3.5 Snake and Prey

The core logic of our game resides within two classes prey.cpp and snake.cpp. Within the prey class, we have a random Coordinate() function that uses rand() function of the C++ library to generate ten random x and 10 random y-coordinates and stores them in an array of random Coordinates[ ]. To ensure that the same coordinates don't keep repeating, we compare the array of previous coordinates with the newly generated coordinates and ran the algorithm until all newly generated coordinates are different from the old one. Also, we have to make sure that the prey doesn't appear exactly in the position of the snakehead. If this is to happen, the game would immediately end thus we also compared the random coordinates with the coordinates of the snakehead.

```
int x = 0;
srand((unsigned int)(time(nullptr)));//seeding unique unsigned integer for random number generation

for(int i = 0; i < 20; i++)
{
    randomCoordinates[i] = 0;
}

for(int i = 0 ; i < 10 ; i++)
{
    x = rand() % 780;
    x = x - (x % 20);
    for(int j = 0 ; j < i ; j++)
    {
        if ( x == (int)randomCoordinates[j] || x == (int)snakeX )
            i--;
    }
    randomCoordinates[i] = (float)x;
}
```

*Fig.3.5.1 Random x coordinate*

From a list of words, a random word is parsed and individual characters are separated, each character is given a unique index which will later be used as texture offset. These texture offsets are used in algebraic equations to obtain respective texture coordinates as shown in Fig.3.5.2.

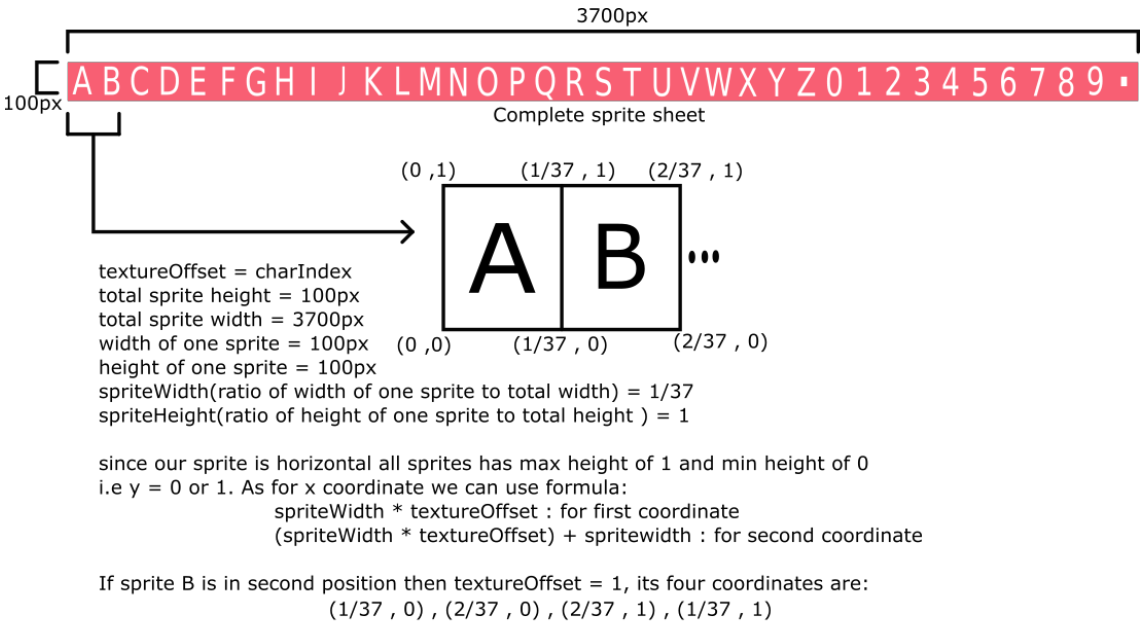


*Fig.3.5.2 Sprite Logic*

The score and stats displayed for the game are also extracted from the same sprite sheet following the flowchart better explains the flow of functions within the class.

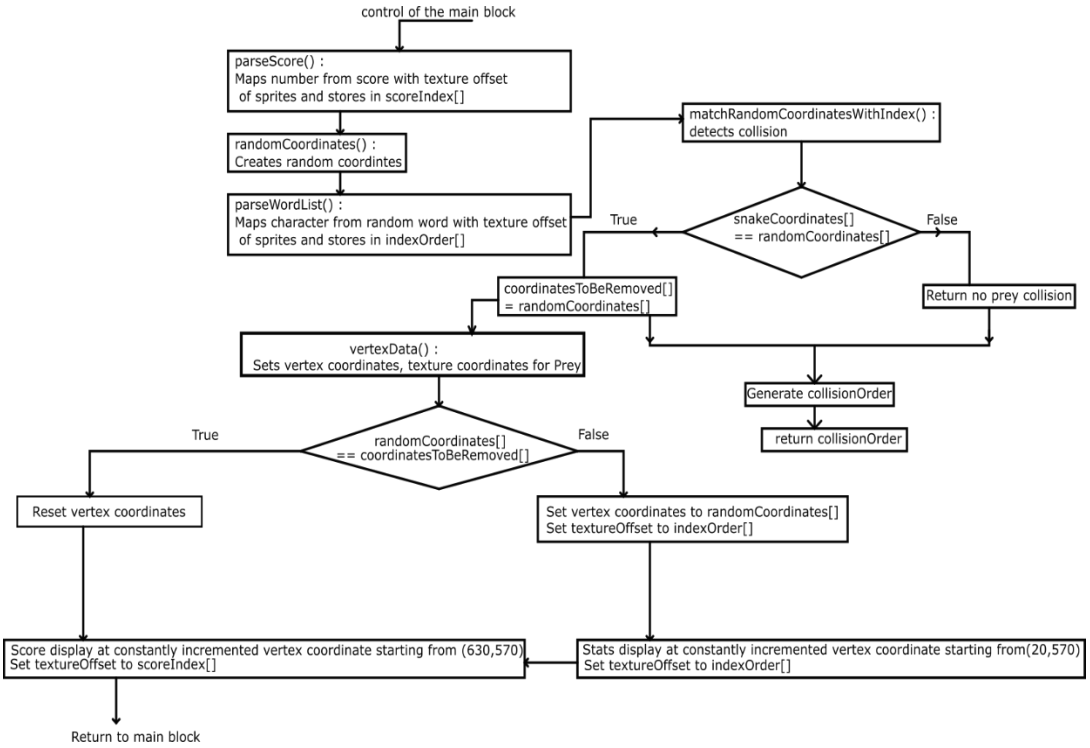


***Fig.3.5.3 Prey Flow Chart***

Vertex buffer for snake is also built-in similar manner except it does not have any texture coordinates and the vertices are incremented or decremented in every frame according to the key input. Key inputs are handled by the GLFWSetKeyCallback() function at the beginning of each frame. We only have to identify which key it is and decide what action we must do for each keypress.

```
static void key_Callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if(key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
    {
        glfwSetWindowShouldClose(window,GLFW_TRUE);
    }

    if (key == GLFW_KEY_W && action == GLFW_PRESS)
    {
        s.setCurrentKey(GLFW_KEY_W);
    }

    if (key == GLFW_KEY_A && action == GLFW_PRESS)
    {
        s.setCurrentKey(GLFW_KEY_A);
    }

    if (key == GLFW_KEY_S && action == GLFW_PRESS)
    {
        s.setCurrentKey(GLFW_KEY_S);
    }

    if (key == GLFW_KEY_D && action == GLFW_PRESS)
    {
        s.setCurrentKey(GLFW_KEY_D);
    }

}
```

***Fig.3.5.4 Key Call back***

The movement of the snake can be achieved by shifting snake body parts to either following or preceding position. It's better explained by the Fig.3.5.5.

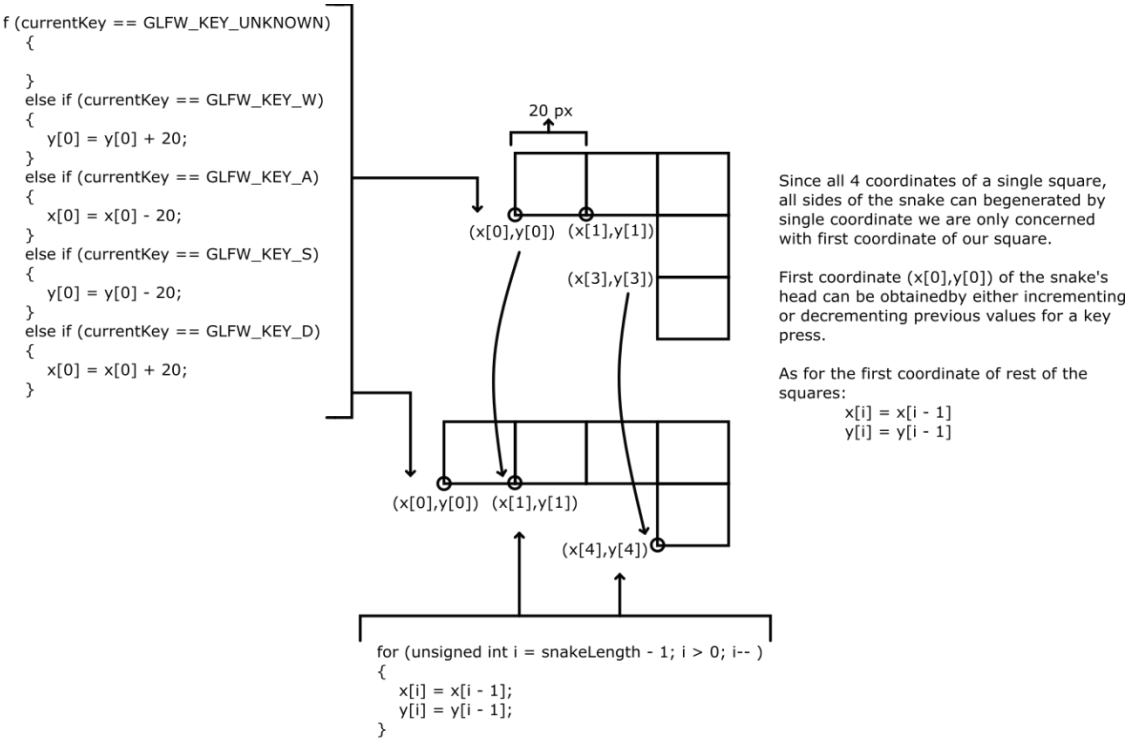


***Fig.3.5.5 Snake movement***

## 3.6 Collision Detection

The current position of the snakehead is passed to the function matchRandomCoordinatesWithIndex( ) where, the random coordinates of the prey are compared with the coordinates of the snakehead, if both are equal it implies a collision has occurred. To determine what reaction needs to happen, the program identifies the collision sequence and creates a flag called CollisionOrder.

This flag is passed to another function preyCollision(), which decides the appropriate reaction for the given CollisionOrder. Along with the CollisionOrder flag, this function also receives two extra flags, selfCollisionOrder and boarderCollisionOrder.

If CollisionOrder == 0

- No collision has occurred

If CollisionOrder == 1

- collision with the correct letter

- If the snakelength < 150 then: snake++
- Remove the prey from the screen

If CollisionOrder == 2

- collision with the last letter of the word.
- If the snakelength < 150 then: snake++
- Remove the prey from the viewport.
- Generate new random coordinates.
- Parse new random letters
- Assign new random coordinates to the new random letters.
- Pass the new coordinates to the prey vertex buffer

If CollisionOrder == 3

- Collision with the wrong letter.
- Call gameOver()
- Generate new random coordinates.
- Parse new random letters
- Assign new random coordinates to the new random letters.
- Pass the new coordinates to the prey vertex buffer

If selfCollisionOrder == 1

- Self collision has occurred
- Call gameOver()
- selfCollisionOrder = 0
- Generate new random coordinates.
- Parse new random letters
- Assign new random coordinates to the new random letters.
- Pass the new coordinates to the prey vertex buffer

If boarderCollisionOrder == 1

- Boarder collision has occurred
- Call gameOver()
- BoarderCollisionOrder = 0

- Generate new random coordinates.
- Parse new random letters
- Assign new random coordinates to the new random letters.
- Pass the new coordinates to the prey vertex buffer

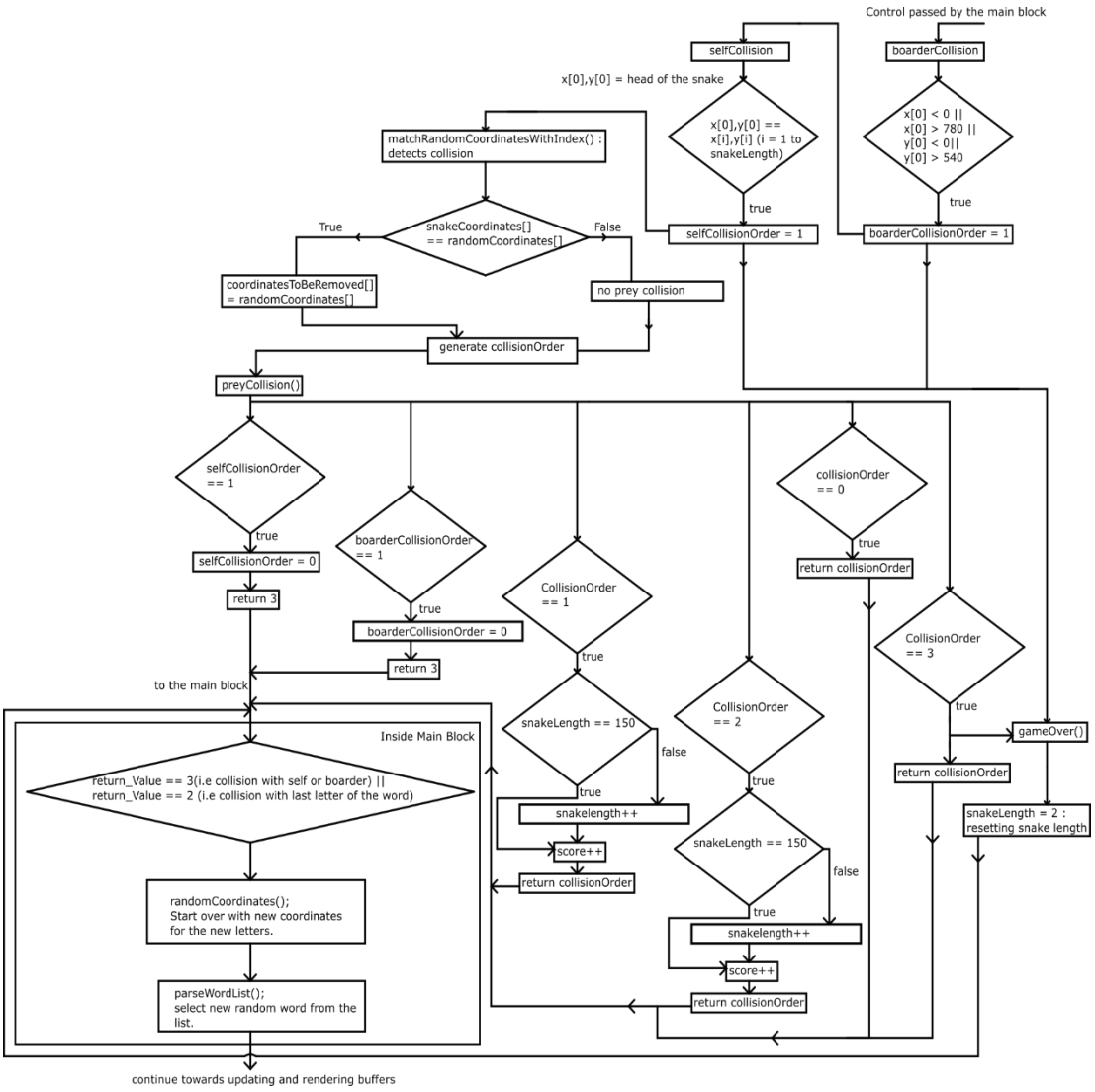


***Fig.3.6.1 Collision Detection***

All the classes and functions abstracted into multiple files are ultimately used by the main block which resides in application.cpp so in order to observe interconnection between these classes we generated a dependency diagram (Fig.3.6.2) of application.cpp using documentation generator; doxygen.

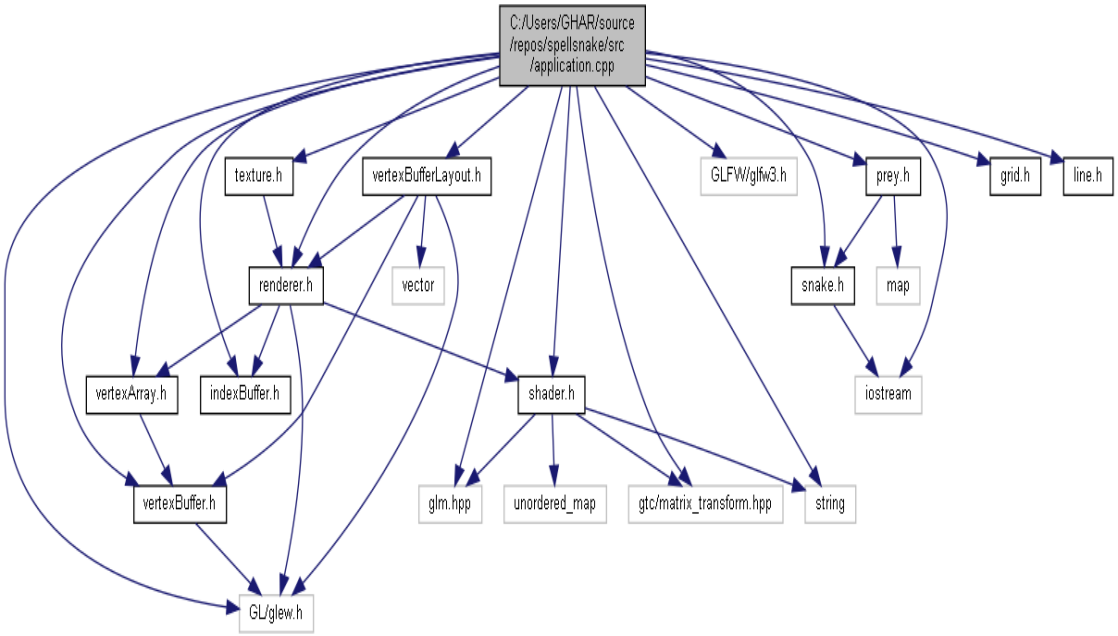


***Fig.3.6.2 Dependencies between header files***

## 3.2. System Requirement Specification

### 3.2.1. User Requirement (Optimal):

- Windows 7 or above OS
- 128 MB or more memory for GPU
- Any CPU supporting windows 7 and above

### 3.2.2. Developer Requirement (Minimal):

- C++, OpenGL 3.3 and above
- Any CPU supporting windows 7 and above
- 256 MB or more memory for GPU
- GLFW, GLM and glew dependencies
- Win32 API
- Any IDE supporting C++ (Visual studio recommended)
- Photoshop equivalent photo editing application for sprite sheet designing

# Chapter 4: Discussion on Achievements

We have always selected our past projects in such a way so as to implement most of our semester learnt knowledge. On top of that the game itself is quite unique and upon research we couldn't find anything like it. The concept of our game itself is an achievement, being able to remake a classic idea into something educational and practicable. Aside of that, designing a rendering pipeline of our own design has been the most technical work we have done till this date and it has made us realise how powerful modern computers are, being able implement such complex calculations to thousands of pixels in mere fraction of seconds. After being familiar with such complex work environment, not only have we gained new skill but a huge boost to our confidence, making us feel like we can tackle any projects.

## 4.1. Features

- Educational and applicable for teaching spellings
- Rendered using minimal draw calls as possible by batching multiple buffers together making the game fast and responsive
- Simple textures and minimal polygon count making the game playable in almost any hardware supporting windows OS
- Word library can be easily changed or expanded without making any changes to the source code

# Chapter 5: Conclusion and Recommendation

Through this project we have managed to create a game not only fun to play but also educational. We also got a chance to explore the flow of graphics developments and design our own rendering mechanism. Through this project we were also able to implement various concepts learned in semester course COMP 342 and explore OpenGL graphics library and further improve our experience on visual studio IDE and C++ programming.

## 5.1. Limitations

Although the game is simple and fun to play and a good visual impression. There are certain limitations to our project.

- Snake length cannot exceed 150 units.
    - 1 unit = 20 px
    - 150 units = 150*20 = 3000 px
- The total size of words is limited to 10 alphabets.
- Can only be run on Windows OS.
- Does not have a leader board or online score sharing platform.

## 5.2. Future Enhancement:

Our game can be developed further to integrate online and co-operation mode. The leader board can be shared and between the players so as to increase the competitiveness between the players. Currently, the game is in its basic stage, it can further be improved by using advanced visual effects. The game at this stage is capable of running only on windows, however this could change with some ad-ons.

# References

Farlex. (n.d.). *hangmanwordgame*. Retrieved from https://hangmanwordgame.com/?fca=1&success=0#/

*GamingHistory*. (1976). Retrieved from https://www.arcade-history.com/?n=blockade&page=detail&id=287

Lee Salzman. (n.d.). *sauerbraten*. Retrieved from http://sauerbraten.org/

Marco Tarini, P. C. (n.d.). *QuteMol*. Retrieved from http://qutemol.sourceforge.net/

# Appendix

The work breakdown and time in weeks required to complete the specific task are shown as in the

Gantt chart below: -

| **Task** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Research and study | ■ | ■ | ■ | | | | | | | | | |
| Graphic Designing | | | | ■ | ■ | ■ | | | | | | |
| Core Programming | | | | | | ■ | ■ | ■ | ■ | ■ | | |
| Program testing | | | | | | | | | | ■ | ■ | |
| Documentation | | | | | | | | | | | ■ | ■ |

***Table 5.1: Gantt Chart***

## **Tasks**:

1. Research and Study: Most of our project development involved learning OpenGL.
2. Graphic Designing: Designing part included the use of OpenGL where C++ was used as a programming language.
3. Core Programming: Core programming involved using C++ to make use of OpenGL and its libraries.
4. Program Testing: Certainly, the project is not a perfection. Internal testing between the team members and classmates was done to minimise the errors.
5. Documentation: The completed project is then documented as a report for the review.